DCA0204, Módulo 2
Ordenação

Daniel Aloise

baseado em slides do prof. Leo Liberti, École Polytechnique, França

DCA, UFRN

Sumário

- Complexidade geral de ordenação
- Mergesort
- Quicksort
- Partição two-way
- Counting sort

O problema de ordenação

- Considere o problema a seguir:

PROBLEMA DE ORDENAÇÃO (PO). Dada uma sequência $s = (s_1, \ldots, s_n)$, encontre uma permutação $\pi \in S_n$ de n símbolos tal que tenhamos a propriedade a seguir:

$$\forall 1 \leq i < j \leq n \; (s_{\pi(i)} \leq s_{\pi(j)}),$$

onde S_n é um conjunto de ordem n

O problema de ordenação

- Considere o problema a seguir:

  PROBLEMA DE ORDENAÇÃO (PO). Dada uma sequência $s = (s_1, \ldots, s_n)$, encontre uma permutação $\pi \in S_n$ de n símbolos tal que tenhamos a propriedade a seguir:

  $$\forall 1 \leq i < j \leq n \left(s_{\pi(i)} \leq s_{\pi(j)}\right),$$

  onde S_n é um conjunto de ordem n
- Em outras palavras, queremos **ordenar** s

O problema de ordenação

- Considere o problema a seguir:

  PROBLEMA DE ORDENAÇÃO (PO). Dada uma sequência $s = (s_1, \ldots, s_n)$, encontre uma permutação $\pi \in S_n$ de n símbolos tal que tenhamos a propriedade a seguir:

  $$\forall 1 \leq i < j \leq n \left(s_{\pi(i)} \leq s_{\pi(j)}\right),$$

  onde S_n é um conjunto de ordem n
- Em outras palavras, queremos **ordenar** s

O *tipo* de s (inteiros, floats, etc.) pode ser importante para desenvolver algoritmos mais eficientes: `mergeSort` e `quickSort` são para tipos genéricos (nenhum conhecimento assumido a priori); em `twoWaySort` sabemos que o tipo é binário

Complexidade de um problema?

- Podemos saber a complexidade do problema de ordenação?
- Lembre: usualmente a complexidade mede o tempo de CPU (número de operações básicas) de um *algoritmo*
- Poderíamos querer saber a complexidade de *pior caso* (para todas as entradas) do *melhor* algoritmo para resolver o problema
- Mas como podemos listar *todos os possíveis algoritmos para um dado problema*?

Esta questão parece difícil de responder

Comparações

- Os elementos cruciais para os algoritmos de ordenação são as **comparações**: dados s_i, s_j, podemos avaliar se $s_i \leq s_j$ é verdadeiro ou falso

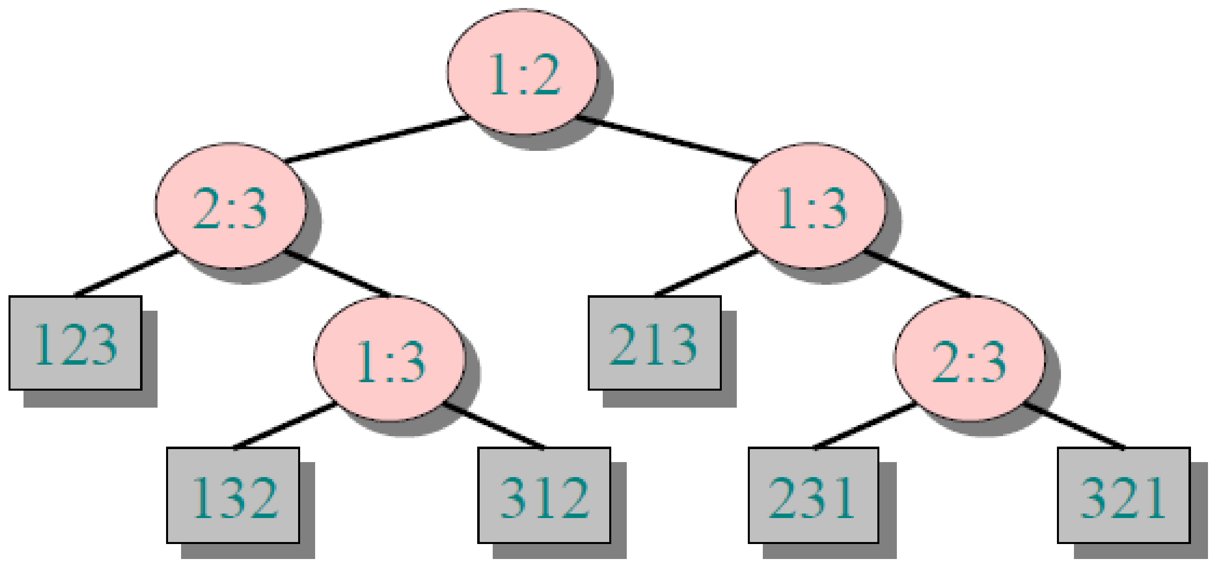
Comparações

- Os elementos cruciais para os algoritmos de ordenação são as **comparações**: dados s_i, s_j, podemos avaliar se $s_i \leq s_j$ é verdadeiro ou falso
- Podemos descrever *qualquer* algoritmo de ordenação por meio de uma **árvore de ordenação**

Comparações

- Os elementos cruciais para os algoritmos de ordenação são as **comparações**: dados s_i, s_j, podemos avaliar se $s_i \leq s_j$ é verdadeiro ou falso
- Podemos descrever *qualquer* algoritmo de ordenação por meio de uma **árvore de ordenação**
- Ex. para ordenar s_1, s_2, s_3, uma possível árvore de ordenação é:

Exemplo de árvore de ordenação para três elementos



Árvores de ordenação

- Cada árvore de ordenação representa uma maneira possível de encadear comparações para ordenar todas as entradas possíveis

Árvores de ordenação

- Cada árvore de ordenação representa uma maneira possível de encadear comparações para ordenar todas as entradas possíveis
- Uma árvore de ordenação fornece todas as saídas possíveis para todas as entradas

Árvores de ordenação

- Cada árvore de ordenação representa uma maneira possível de encadear comparações para ordenar todas as entradas possíveis
- Uma árvore de ordenação fornece todas as saídas possíveis para todas as entradas
- Qualquer algoritmo de ordenação corresponde a uma árvore de ordenação particular

Árvores de ordenação

- Cada árvore de ordenação representa uma maneira possível de encadear comparações para ordenar todas as entradas possíveis
- Uma árvore de ordenação fornece todas as saídas possíveis para todas as entradas
- Qualquer algoritmo de ordenação corresponde a uma árvore de ordenação particular
- O número de algoritmos de ordenação é no máximo igual ao número de árvores de ordenação

Árvores de ordenação

- Cada árvore de ordenação representa uma maneira possível de encadear comparações para ordenar todas as entradas possíveis
- Uma árvore de ordenação fornece todas as saídas possíveis para todas as entradas
- Qualquer algoritmo de ordenação corresponde a uma árvore de ordenação particular
- O número de algoritmos de ordenação é no máximo igual ao número de árvores de ordenação
- Podemos usar árvores de ordenação para expressar a ideia do *melhor algoritmo de ordenação possível*

Melhor complexidade de pior caso

- Seja $\mathbb{T}_n$ o conjunto de todas as árvores de ordenação para sequências de tamanho n

Melhor complexidade de pior caso

- Seja $\mathbb{T}_n$ o conjunto de todas as árvores de ordenação para sequências de tamanho n
- Entradas diferentes levam a diferentes permutações de ordenação nos *nós folha* de cada árvore de ordenação

Melhor complexidade de pior caso

- Seja $\mathbb{T}_n$ o conjunto de todas as árvores de ordenação para sequências de tamanho n
- Entradas diferentes levam a diferentes permutações de ordenação nos *nós folha* de cada árvore de ordenação
- Para uma árvore de ordenação $T \in \mathbb{T}_n$ e um $\pi \in S_n$ denotamos por $\ell(T, \pi)$ o tamanho do caminho em T da raiz até a folha contendo π

Melhor complexidade de pior caso

- Seja $\mathbb{T}_n$ o conjunto de todas as árvores de ordenação para sequências de tamanho n
- Entradas diferentes levam a diferentes permutações de ordenação nos *nós folha* de cada árvore de ordenação
- Para uma árvore de ordenação $T \in \mathbb{T}_n$ e um $\pi \in S_n$ denotamos por $\ell(T, \pi)$ o tamanho do caminho em T da raiz até a folha contendo π
- A melhor complexidade de pior caso é, para cada $n \geq 0$:

$$B_n = \min_{T \in \mathbb{T}_n} \max_{\pi \in S_n} \ell(T, \pi).$$

Melhor complexidade de pior caso

- Seja $\mathbb{T}_n$ o conjunto de todas as árvores de ordenação para sequências de tamanho n
- Entradas diferentes levam a diferentes permutações de ordenação nos *nós folha* de cada árvore de ordenação
- Para uma árvore de ordenação $T \in \mathbb{T}_n$ e um $\pi \in S_n$ denotamos por $\ell(T, \pi)$ o tamanho do caminho em T da raiz até a folha contendo π
- A melhor complexidade de pior caso é, para cada $n \geq 0$:

$$B_n = \min_{T \in \mathbb{T}_n} \max_{\pi \in S_n} \ell(T, \pi).$$

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T
- **Profundidade da árvore**: caminho de comprimento mais longo entre uma raiz e uma folha em uma árvore

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T
- **Profundidade da árvore**: caminho de comprimento mais longo entre uma raiz e uma folha em uma árvore
- Uma árvore binária T com profundidade limitada por k tem $|V(T)| \leq 2^k$

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T
- **Profundidade da árvore**: caminho de comprimento mais longo entre uma raiz e uma folha em uma árvore
- Uma árvore binária T com profundidade limitada por k tem $|V(T)| \leq 2^k$
- $\Rightarrow$ A árvore de ordenação T^* do melhor algoritmo tem $|V(T^*)| \leq 2^{B_n}$

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T
- **Profundidade da árvore**: caminho de comprimento mais longo entre uma raiz e uma folha em uma árvore
- Uma árvore binária T com profundidade limitada por k tem $|V(T)| \leq 2^k$
- $\Rightarrow$ A árvore de ordenação T^* do melhor algoritmo tem $|V(T^*)| \leq 2^{B_n}$
- $\forall T \in \mathbb{T}_n$, cada $\pi \in S_n$ aparece em uma folha de T

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T
- **Profundidade da árvore**: caminho de comprimento mais longo entre uma raiz e uma folha em uma árvore
- Uma árvore binária T com profundidade limitada por k tem $|V(T)| \leq 2^k$
- $\Rightarrow$ A árvore de ordenação T^* do melhor algoritmo tem $|V(T^*)| \leq 2^{B_n}$
- $\forall T \in \mathbb{T}_n$, cada $\pi \in S_n$ aparece em uma folha de T
- $\Rightarrow$ Qualquer $T \in \mathbb{T}_n$ tem pelo menos $n!$ folhas, i.e. $|V(T)| \geq n!$

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T
- **Profundidade da árvore**: caminho de comprimento mais longo entre uma raiz e uma folha em uma árvore
- Uma árvore binária T com profundidade limitada por k tem $|V(T)| \leq 2^k$
- $\Rightarrow$ A árvore de ordenação T^* do melhor algoritmo tem $|V(T^*)| \leq 2^{B_n}$
- $\forall T \in \mathbb{T}_n$, cada $\pi \in S_n$ aparece em uma folha de T
- $\Rightarrow$ Qualquer $T \in \mathbb{T}_n$ tem pelo menos $n!$ folhas, i.e. $|V(T)| \geq n!$
- Portanto, $n! \leq 2^{B_n}$, o que implica $B_n \geq \lceil \log n! \rceil$

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T
- **Profundidade da árvore**: caminho de comprimento mais longo entre uma raiz e uma folha em uma árvore
- Uma árvore binária T com profundidade limitada por k tem $|V(T)| \leq 2^k$
- $\Rightarrow$ A árvore de ordenação T^* do melhor algoritmo tem $|V(T^*)| \leq 2^{B_n}$
- $\forall T \in \mathbb{T}_n$, cada $\pi \in S_n$ aparece em uma folha de T
- $\Rightarrow$ Qualquer $T \in \mathbb{T}_n$ tem pelo menos $n!$ folhas, i.e. $|V(T)| \geq n!$
- Portanto, $n! \leq 2^{B_n}$, o que implica $B_n \geq \lceil \log n! \rceil$
- Pela aprox. de Stirling, $\log n! = n \log n - \frac{1}{\ln 2} n + O(\log n)$

A complexidade da ordenação

- Para qualquer árvore T, seja $|V(T)|$ o número de nós de T
- **Profundidade da árvore**: caminho de comprimento mais longo entre uma raiz e uma folha em uma árvore
- Uma árvore binária T com profundidade limitada por k tem $|V(T)| \leq 2^k$
- $\Rightarrow$ A árvore de ordenação T^* do melhor algoritmo tem $|V(T^*)| \leq 2^{B_n}$
- $\forall T \in \mathbb{T}_n$, cada $\pi \in S_n$ aparece em uma folha de T
- $\Rightarrow$ Qualquer $T \in \mathbb{T}_n$ tem pelo menos $n!$ folhas, i.e. $|V(T)| \geq n!$
- Portanto, $n! \leq 2^{B_n}$, o que implica $B_n \geq \lceil \log n! \rceil$
- Pela aprox. de Stirling, $\log n! = n \log n - \frac{1}{\ln 2} n + O(\log n)$
- $\Rightarrow$ B_n é limitado inferiormente por uma função proporcional a $n \log n$ (dizemos que B_n é $\Omega(n \log n)$)

Resultado mágico de hoje: primeira parte

Complexidade da ordenação: $\Omega(n \log n)$

Algoritmos simples de ordenação

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2), \varnothing$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$\rightarrow (3, 4, \boxed{2}), (1)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$\rightarrow (\boxed{3}, 4), (1, 2)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$\rightarrow (\boxed{4}), (1, 2, 3)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$\rightarrow (1, 2, 3, 4)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2) \rightarrow (3, 4, \boxed{2}), (1) \rightarrow (\boxed{3}, 4), (1, 2) \rightarrow (\boxed{4}), (1, 2, 3) \rightarrow (1, 2, 3, 4)$$

- e o **insertion sort**, onde você insere o próximo elemento de s em sua posição correta na sequência ordenada

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2) \rightarrow (3, 4, \boxed{2}), (1) \rightarrow (\boxed{3}, 4), (1, 2) \rightarrow (\boxed{4}), (1, 2, 3) \rightarrow (1, 2, 3, 4)$$

- e o **insertion sort**, onde você insere o próximo elemento de s em sua posição correta na sequência ordenada

$$(\boxed{3}, 1, 4, 2)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2) \to (3, 4, \boxed{2}), (1) \to (\boxed{3}, 4), (1, 2) \to (\boxed{4}), (1, 2, 3) \to (1, 2, 3, 4)$$

- e o **insertion sort**, onde você insere o próximo elemento de s em sua posição correta na sequência ordenada

$$\to (\boxed{1}, 4, 2), (3)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2) \to (3, 4, \boxed{2}), (1) \to (\boxed{3}, 4), (1, 2) \to (\boxed{4}), (1, 2, 3) \to (1, 2, 3, 4)$$

- e o **insertion sort**, onde você insere o próximo elemento de s em sua posição correta na sequência ordenada

$$\to (\boxed{4}, 2), (1, 3)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2) \rightarrow (3, 4, \boxed{2}), (1) \rightarrow (\boxed{3}, 4), (1, 2) \rightarrow (\boxed{4}), (1, 2, 3) \rightarrow (1, 2, 3, 4)$$

- e o **insertion sort**, onde você insere o próximo elemento de s em sua posição correta na sequência ordenada

$$\rightarrow (\boxed{2}), (1, 3, 4)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2) \rightarrow (3, 4, \boxed{2}), (1) \rightarrow (\boxed{3}, 4), (1, 2) \rightarrow (\boxed{4}), (1, 2, 3) \rightarrow (1, 2, 3, 4)$$

- e o **insertion sort**, onde você insere o próximo elemento de s em sua posição correta na sequência ordenada

$$\rightarrow (1, 2, 3, 4)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2) \rightarrow (3, 4, \boxed{2}), (1) \rightarrow (\boxed{3}, 4), (1, 2) \rightarrow (\boxed{4}), (1, 2, 3) \rightarrow (1, 2, 3, 4)$$

- e o **insertion sort**, onde você insere o próximo elemento de s em sua posição correta na sequência ordenada

$$(\boxed{3}, 1, 4, 2) \rightarrow (\boxed{1}, 4, 2), (3) \rightarrow (\boxed{4}, 2), (1, 3) \rightarrow (\boxed{2}), (1, 3, 4) \rightarrow (1, 2, 3, 4)$$

Algoritmos simples de ordenação

- Eu vou economizar o seu tempo de aprender *todos* os diversos tipos existentes de algoritmos de ordenação
- Permita-me apenas mencionar o **selection sort**, onde você seleciona repetidamente o *menor* elemento de s,

$$(3, \boxed{1}, 4, 2) \rightarrow (3, 4, \boxed{2}), (1) \rightarrow (\boxed{3}, 4), (1, 2) \rightarrow (\boxed{4}), (1, 2, 3) \rightarrow (1, 2, 3, 4)$$

- e o **insertion sort**, onde você insere o próximo elemento de s em sua posição correta na sequência ordenada

$$(\boxed{3}, 1, 4, 2) \rightarrow (\boxed{1}, 4, 2), (3) \rightarrow (\boxed{4}, 2), (1, 3) \rightarrow (\boxed{2}), (1, 3, 4) \rightarrow (1, 2, 3, 4)$$

- Ambos são $O(n^2)$

Mergesort

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se* $|s| \leq 1$ *então* s *já está ordenado por definição*

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se* $|s| \leq 1$ *então* s *já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se* $|s| \leq 1$ *então* s *já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{array}{c} (2,3,5,6) \\ (\boxed{1},3,4,9) \end{array} \rightarrow \varnothing$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se $|s| \leq 1$ então s já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{array}{c} (\boxed{2},3,5,6) \\ (1,3,4,9) \end{array} \rightarrow (1)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se* $|s| \leq 1$ *então* s *já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{array}{c}(2,\boxed{3},5,6)\\(1,3,4,9)\end{array} \rightarrow (1, 2)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se $|s| \leq 1$ então s já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{array}{c} (2,3,5,6) \\ (1,\boxed{3},4,9) \end{array} \rightarrow (1, 2, 3)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se $|s| \leq 1$ então s já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{array}{c} (2,3,5,6) \\ (1,3,\boxed{4},9) \end{array} \rightarrow (1, 2, 3, 3)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se $|s| \leq 1$ então s já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{array}{c} (2,3,\boxed{5},6) \\ (1,3,4,9) \end{array} \rightarrow (1, 2, 3, 3, 4)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se* $|s| \leq 1$ *então* s *já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{matrix} (2,3,5,\boxed{6}) \\ (1,3,4,9) \end{matrix} \rightarrow (1, 2, 3, 3, 4, 5)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se $|s| \leq 1$ então s já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{matrix} (2,3,5,6) \\ (1,3,4,\boxed{9}) \end{matrix} \rightarrow (1, 2, 3, 3, 4, 5, 6)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se $|s| \leq 1$ então s já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{matrix}(2,3,5,6)\\(1,3,4,9)\end{matrix} \rightarrow (1, 2, 3, 3, 4, 5, 6, 9) = \bar{s}$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Divida s ao meio: a primeira metade é $s' = (5, 3, 6, 2)$ e a segunda é $s'' = (1, 9, 4, 3)$
- Ordene s', s'': uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base é quando $|s| \leq 1$
- *Se $|s| \leq 1$ então s já está ordenado por definição*
- Tome $s' = (2, 3, 5, 6)$ e $s'' = (1, 3, 4, 9)$
- Junte s', s'' em uma sequência ordenada $\bar{s}$:

$$\begin{matrix}(2,3,5,6)\\(1,3,4,9)\end{matrix} \rightarrow (1, 2, 3, 3, 4, 5, 6, 9) = \bar{s}$$

- Retorne $\bar{s}$

Juntar = *Merge*

- `merge`(s', s''): junta duas sequências ordenadas s', s'' em uma sequência ordenada contendo todos os elementos em s', s''

Juntar = *Merge*

- `merge`(s', s''): junta duas sequências ordenadas s', s'' em uma sequência ordenada contendo todos os elementos em s', s''
- Uma vez que s', s'' estão ambos já ordenados, juntá-los de modo que o resultado seja ordenado é eficiente
  - Leia os primeiros (e menores) elementos de s', s'': $O(1)$
  - Compare estes dois elementos: $O(1)$
  - Existem $|s|$ elementos para processar: $O(n)$

Juntar = *Merge*

- `merge`(s', s''): junta duas sequências ordenadas s', s'' em uma sequência ordenada contendo todos os elementos em s', s''
- Uma vez que s', s'' estão ambos já ordenados, juntá-los de modo que o resultado seja ordenado é eficiente
  - Leia os primeiros (e menores) elementos de s', s'': $O(1)$
  - Compare estes dois elementos: $O(1)$
  - Existem $|s|$ elementos para processar: $O(n)$
- Você pode implementar isto usando listas: se s' está vazia retorne s'', se s'' está vazia retorne s', e caso contrário compare os primeiros elementos de ambas e escolha o menor elemento

Algoritmo recursivo

- $\texttt{mergeSort}(s)$ {

```
if |s| ≤ 1 then
    return s;
else
    m = ⌊|s|/2⌋;
    s' = mergeSort(e_1, ..., e_m);
    s'' = mergeSort(e_{m+1}, ..., e_n);
    return merge(s', s'');
end if
```

}

Algoritmo tem complexidade $O(n \log n)$ [Cormen et a

Resultado mágico de hoje: segunda parte

Complexidade da ordenação: $\Theta(n \log n)$

Uma função é $\Theta(g(n))$ se ela é tanto $O(g(n))$ quanto $\Omega(g(n))$

Quicksort

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, \boxed{3}, 6, 2, 1, 9, 4, 3) \rightarrow \emptyset, \emptyset$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \rightarrow (3), \varnothing$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, \boxed{6}, 2, 1, 9, 4, 3) \rightarrow (3), \varnothing$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \rightarrow (3), (6)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, \boxed{2}, 1, 9, 4, 3) \rightarrow (3), (6)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \to (3, 2), (6)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, \boxed{1}, 9, 4, 3) \rightarrow (3, 2), (6)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \rightarrow (3, 2, 1), (6)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, \boxed{9}, 4, 3) \rightarrow (3, 2, 1), (6)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \to (3, 2, 1), (6, 9)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, \boxed{4}, 3) \rightarrow (3, 2, 1), (6, 9)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \rightarrow (3, 2, 1, 4), (6, 9)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, \boxed{3}) \rightarrow (3, 2, 1, 4), (6, 9)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \rightarrow (3, 2, 1, 4, 3), (6, 9)$$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \to (3, 2, 1, 4, 3), (6, 9)$$

- Ordene $s' = (3, 2, 1, 4, 3)$ e $s'' = (6, 9)$: uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base $|s| \leq 1$

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \rightarrow (3, 2, 1, 4, 3), (6, 9)$$

- Ordene $s' = (3, 2, 1, 4, 3)$ e $s'' = (6, 9)$: uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base $|s| \leq 1$
- Atualize s para (s', p, s'')

Dividir-para-conquistar

- Seja $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Escolha um valor *pivô* $p = s_1 = 5$ (nenhuma razão particular para escolha de s_1)
- Particione $(s_2, \ldots, s_n)$ em s' (elementos menores do que p) e s'' (elementos maiores ou iguais a p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \rightarrow (3, 2, 1, 4, 3), (6, 9)$$

- Ordene $s' = (3, 2, 1, 4, 3)$ e $s'' = (6, 9)$: uma vez que $|s'| < |s|$ e $|s''| < |s|$ podemos usar recursão; caso base $|s| \leq 1$
- Atualize s para (s', p, s'')

Atenção: no `mergeSort`, nós fazemos a recursão *primeiro*, e então trabalhamos nas subsequências *depois*. In `quickSort`, nós trabalhamos nas subsequências *primeiro*, e então fazemos recursão nelas *depois*.

Particionamento

- `partition`(s): produz duas subsequências s', s'' de $(s_2, \ldots, s_n)$ tal que:
  - $s' = (s_i \mid i \neq 1 \wedge \; s_i < s_1)$
  - $s'' = (s_i \mid i \neq 1 \wedge \; s_i \geq s_1)$

Particionamento

- `partition`(s): produz duas subsequências s', s'' de $(s_2, \ldots, s_n)$ tal que:
  - $s' = (s_i \mid i \neq 1 \wedge \; s_i < s_1)$
  - $s'' = (s_i \mid i \neq 1 \wedge \; s_i \geq s_1)$
- Percorra s: se $s_i < s_1$ coloque s_i em s', caso contrário coloque-o em s''

Particionamento

- `partition`(s): produz duas subsequências s', s'' de $(s_2, \ldots, s_n)$ tal que:
  - $s' = (s_i \mid i \neq 1 \wedge \ s_i < s_1)$
  - $s'' = (s_i \mid i \neq 1 \wedge \ s_i \geq s_1)$
- Percorra s: se $s_i < s_1$ coloque s_i em s', caso contrário coloque-o em s''
- Existem $|s| - 1$ elementos para processar: $O(n)$

Particionamento

- `partition`(s): produz duas subsequências s', s'' de $(s_2, \ldots, s_n)$ tal que:
  - $s' = (s_i \mid i \neq 1 \wedge \; s_i < s_1)$
  - $s'' = (s_i \mid i \neq 1 \wedge \; s_i \geq s_1)$
- Percorra s: se $s_i < s_1$ coloque s_i em s', caso contrário coloque-o em s''
- Existem $|s| - 1$ elementos para processar: $O(n)$
- Você pode implementar isto usando arrays; além disso, se você usar uma função `swap` tal que, dados i, j, troca s_i com s_j em s, você não precisa criar nenhum array temporário : *você pode atualizar s "in place"*

Algoritmo recursivo

- $\texttt{quickSort}(s)$ {

  if $|s| \leq 1$ **then**
      return ;
  else
      $(s', s'') = \texttt{partition}(s)$;
      $\texttt{quickSort}(s')$;
      $\texttt{quickSort}(s'')$;
      $s \leftarrow (s', s_1, s'')$;
  end if

  }

Complexidade

Complexidade de pior caso: $O(n^2)$

Complexidade de caso médio: $O(n \log n)$

Muito rápido na prática

Worst-case complexity

- Considere a entrada $(n, n-1, \ldots, 1)$ com pivô s_1

Worst-case complexity

- Considere a entrada $(n, n-1, \ldots, 1)$ com pivô s_1
- Nível de recursão 1: $p = n$, $s' = (n-1, \ldots, 1)$, $s'' = \emptyset$

Worst-case complexity

- Considere a entrada $(n, n-1, \ldots, 1)$ com pivô s_1
- Nível de recursão 1: $p = n$, $s' = (n-1, \ldots, 1)$, $s'' = \varnothing$
- Nível de recursão 2: $p = n-1$, $s' = (n-2, \ldots, 1)$, $s'' = \varnothing$

Worst-case complexity

- Considere a entrada $(n, n-1, \ldots, 1)$ com pivô s_1
- Nível de recursão 1: $p = n$, $s' = (n-1, \ldots, 1)$, $s'' = \emptyset$
- Nível de recursão 2: $p = n-1$, $s' = (n-2, \ldots, 1)$, $s'' = \emptyset$
- E assim por diante, até $p = 1$ (caso base)

Worst-case complexity

- Considere a entrada $(n, n-1, \ldots, 1)$ com pivô s_1
- Nível de recursão 1: $p = n$, $s' = (n-1, \ldots, 1)$, $s'' = \varnothing$
- Nível de recursão 2: $p = n-1$, $s' = (n-2, \ldots, 1)$, $s'' = \varnothing$
- E assim por diante, até $p = 1$ (caso base)
- Cada chamada à função `partition` leva tempo $O(n)$

Worst-case complexity

- Considere a entrada $(n, n-1, \ldots, 1)$ com pivô s_1
- Nível de recursão 1: $p = n$, $s' = (n-1, \ldots, 1)$, $s'' = \varnothing$
- Nível de recursão 2: $p = n-1$, $s' = (n-2, \ldots, 1)$, $s'' = \varnothing$
- E assim por diante, até $p = 1$ (caso base)
- Cada chamada à função `partition` leva tempo $O(n)$
- Obtem-se $O(n^2)$

Partição 2-Way

Definição através de exemplo

Entada: $(1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
Saída desejada: $(0, 0, 0, 0, 0, 1, 1, 1, 1, 1)$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita
- Repita até que os contadores se tornem iguais

$$(\boxed{1}, 0, 0, 1, 1, 0, 0, \boxed{0}, 1, 1)$$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita
- Repita até que os contadores se tornem iguais

$$(\mathbf{0}, 0, 0, 1, 1, 0, 0, \mathbf{1}, 1, 1)$$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita
- Repita até que os contadores se tornem iguais

$$(0, 0, 0, \boxed{1}, 1, 0, \boxed{0}, 1, 1, 1)$$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita
- Repita até que os contadores se tornem iguais

$$(0, 0, 0, \mathbf{0}, 1, 0, \mathbf{1}, 1, 1, 1)$$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita
- Repita até que os contadores se tornem iguais

$$(0, 0, 0, 0, \boxed{1}, \boxed{0}, 1, 1, 1, 1)$$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita
- Repita até que os contadores se tornem iguais

$$(0, 0, 0, 0, \mathbf{0}, \mathbf{1}, 1, 1, 1, 1)$$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita
- Repita até que os contadores se tornem iguais

$$(0, 0, 0, 0, 0, 1, 1, 1, 1, 1)$$

Trocas iterativas

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Encontre o 1 mais à esquerda e o 0 mais à direita (estes estão fora do lugar)
- Troque eles
- Aumente o contador mais à esquerda, diminua o contador mais à direita
- Repita até que os contadores se tornem iguais

$$(1, 0, 0, 1, 1, 0, 0, 0, 1, 1) \rightarrow (\mathbf{0}, 0, 0, 1, 1, 0, 0, \mathbf{1}, 1, 1) \rightarrow$$
$$(0, 0, 0, \mathbf{0}, 1, 0, \mathbf{1}, 1, 1, 1) \rightarrow (0, 0, 0, 0, \mathbf{0}, \mathbf{1}, 1, 1, 1, 1)$$

O algoritmo

$i = 0$; $j = n - 1$;
while $i \leq j$ **do**
  if $s_i = 0$ **then**
    $i \leftarrow i + 1$;
  else if $s_j = 1$ **then**
    $j \leftarrow j - 1$;
  else
    swap(s, i, j);
    $i \leftarrow i + 1$;
    $j \leftarrow j - 1$;
  end if
end while

Complexidade de pior caso

- Ocorre com entrada $(1, \ldots, 1, 0, \ldots, 0)$ onde o número de 1's é praticamente o mesmo que o número de 0's
- Requer $\lfloor \frac{n}{2} \rfloor$ trocas
- Pior caso: $O(n)$

Um paradoxo?

- No início, provamos que o problema de ordenação tem complexidade $\Theta(n \log n)$

Um paradoxo?

- No início, provamos que o problema de ordenação tem complexidade $\Theta(n \log n)$
- Mas o particionamento 2-way requer tempo $O(n)$

Um paradoxo?

- No início, provamos que o problema de ordenação tem complexidade $\Theta(n \log n)$
- Mas o particionamento 2-way requer tempo $O(n)$
- Contradição? Paradoxo?

Um paradoxo?

- No início, provamos que o problema de ordenação tem complexidade $\Theta(n \log n)$
- Mas o particionamento 2-way requer tempo $O(n)$
- Contradição? Paradoxo?
- Apenas aparente: nosso teorema inicial considerava as seguintes hipóteses:
  - nenhum conhecimento a priori sobre o tipo dos dados de entrada
  - se referia apenas a **algoritmos baseados em comparações**

Um paradoxo?

- No início, provamos que o problema de ordenação tem complexidade $\Theta(n \log n)$
- Mas o particionamento 2-way requer tempo $O(n)$
- Contradição? Paradoxo?
- Apenas aparente: nosso teorema inicial considerava as seguintes hipóteses:
  - nenhum conhecimento a priori sobre o tipo dos dados de entrada
  - se referia apenas a **algoritmos baseados em comparações**
- Nenhuma desta hipóteses é válida para o particionamento 2-way
  - nós sabemos que a entrada é uma sequência de binários
  - o algoritmo não usa comparações

Counting Sort

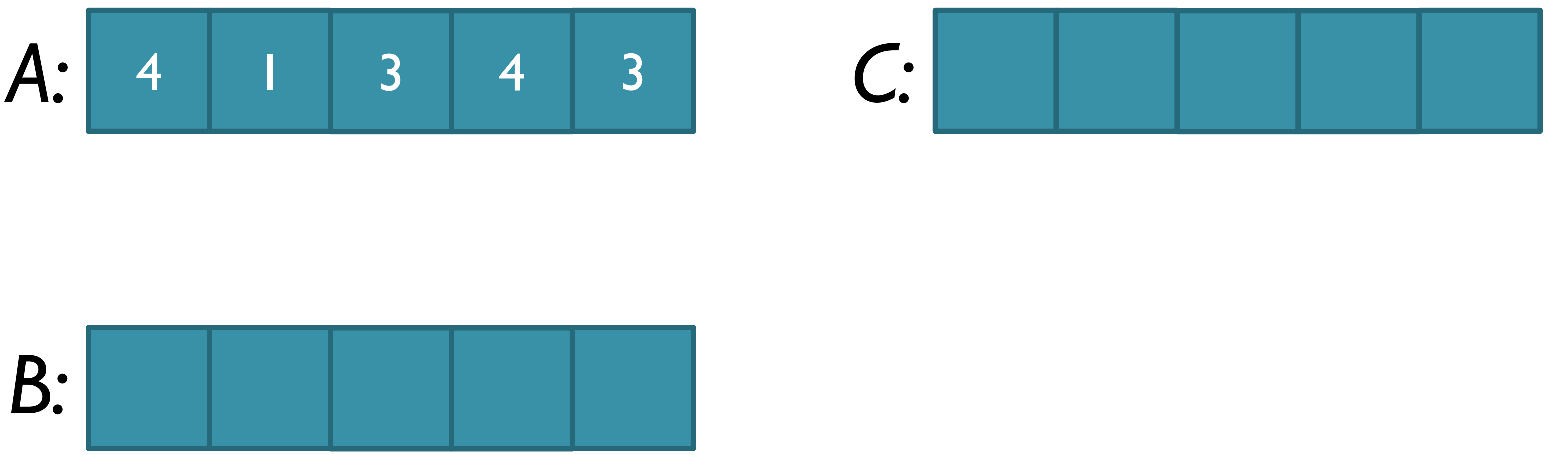
Counting Sort

- Não faz comparações entre os elementos
- Entrada: vetor de n elementos inteiros positivos, para os quais sabemos o maior elemento k
- Saída: vetor ordenado
- Vetor auxiliar: C[1...k]

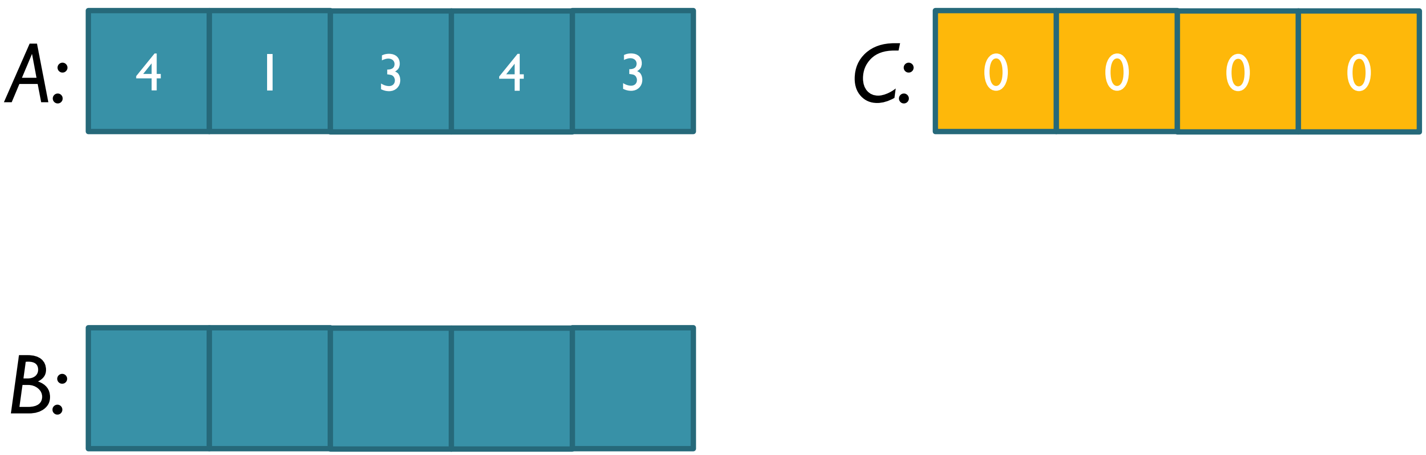
Pseudocódigo

for $i \leftarrow 1$ **to** k
    do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** n
    do $C[A[j]] \leftarrow C[A[j]] + 1$
for $i \leftarrow 2$ **to** k
    do $C[i] \leftarrow C[i] + C[i\text{–}1]$
for $j \leftarrow n$ **downto** 1
    do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$

Exemplo



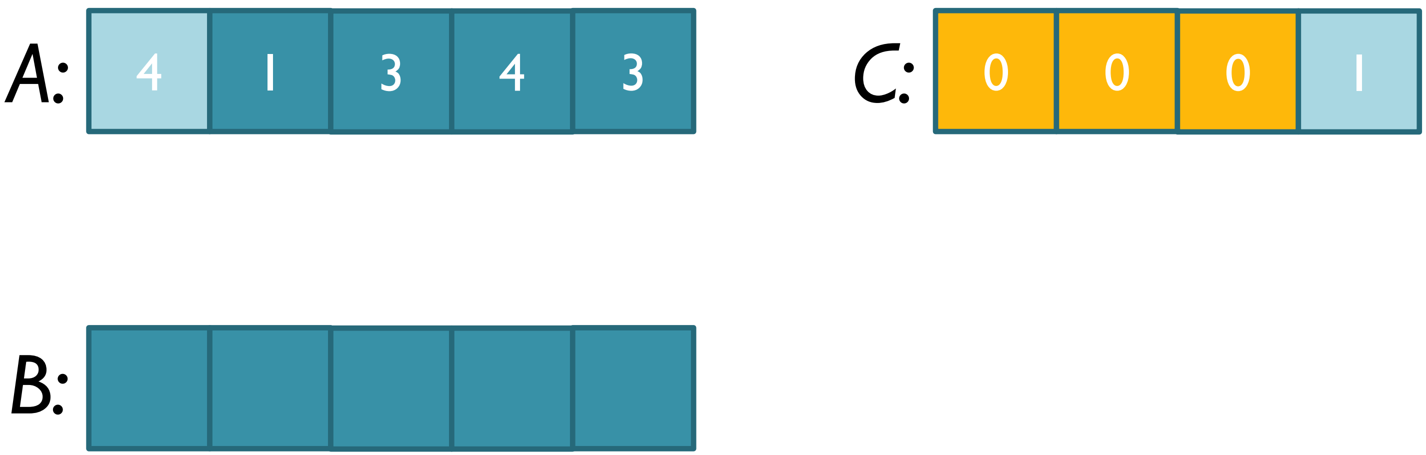
Loop 1



for $i \leftarrow 1$ **to** k

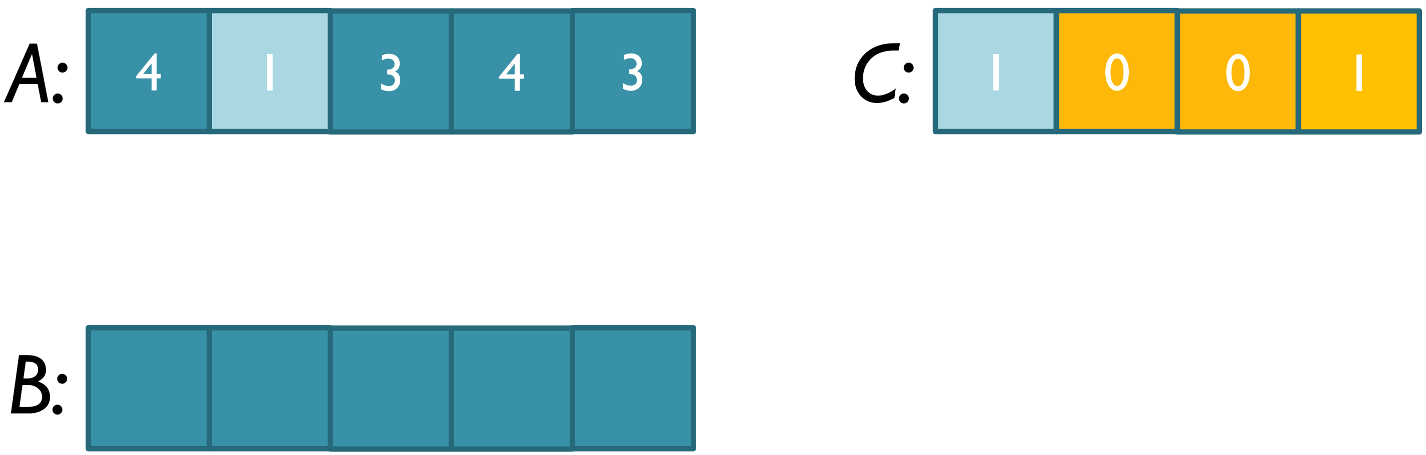
  do $C[i] \leftarrow 0$

Loop 2



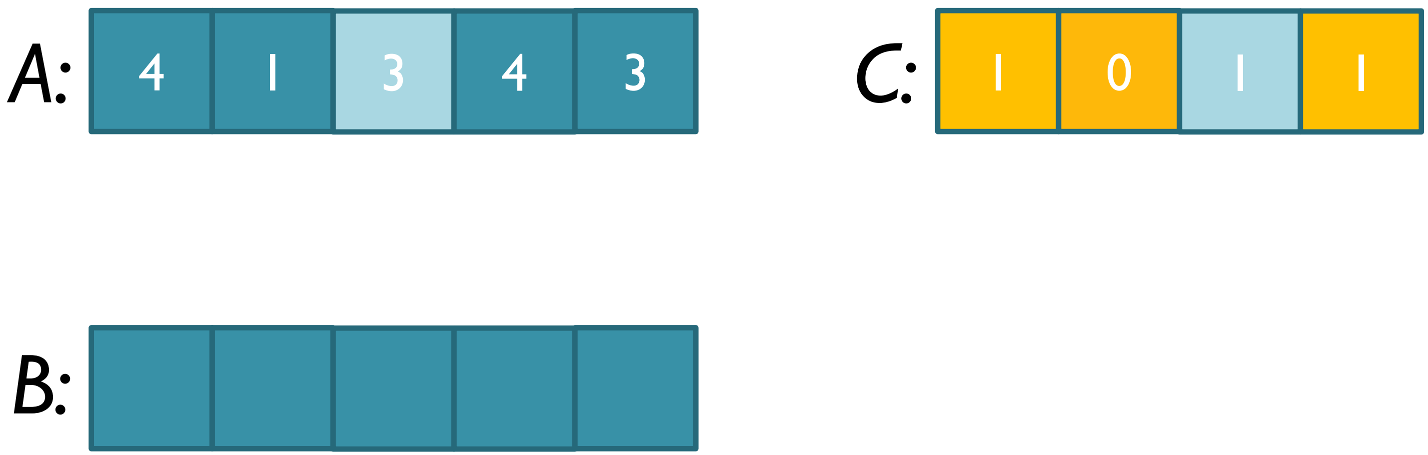
for $j \leftarrow 1$ **to** n
 do $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 2



for $j \leftarrow 1$ **to** n
    do $C[A[j]] \leftarrow C[A[j]] + 1$

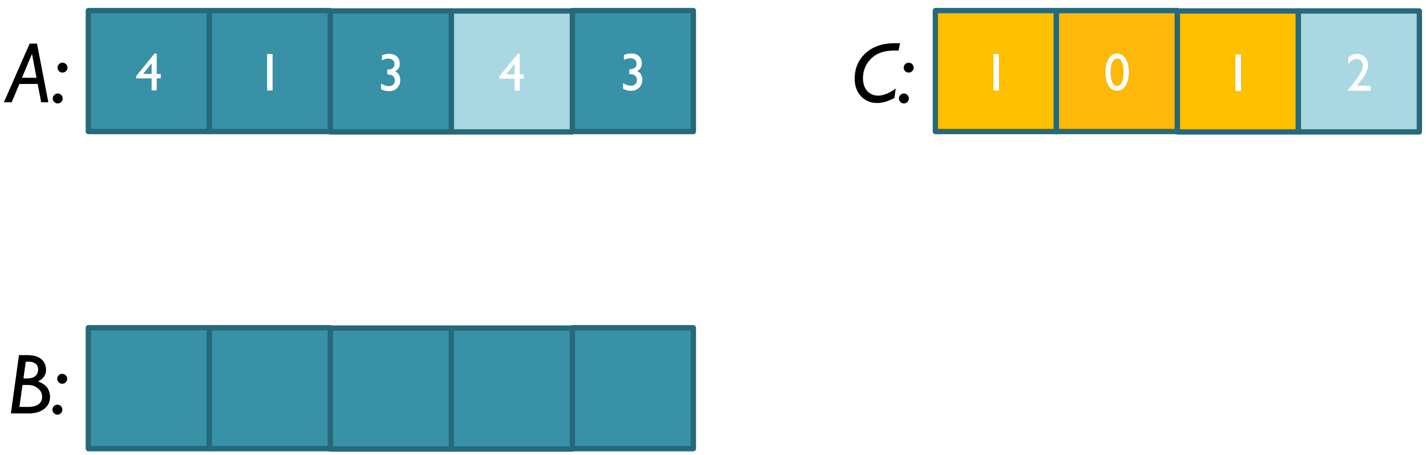
Loop 2



for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 2



for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 2

A:

4	1	3	4	3

C:

1	0	2	2

B:

for $j \leftarrow 1$ **to** n

  do $C[A[j]] \leftarrow C[A[j]] + 1$

Loop 3

A:

4	1	3	4	3

C:

1	0	2	2

B:

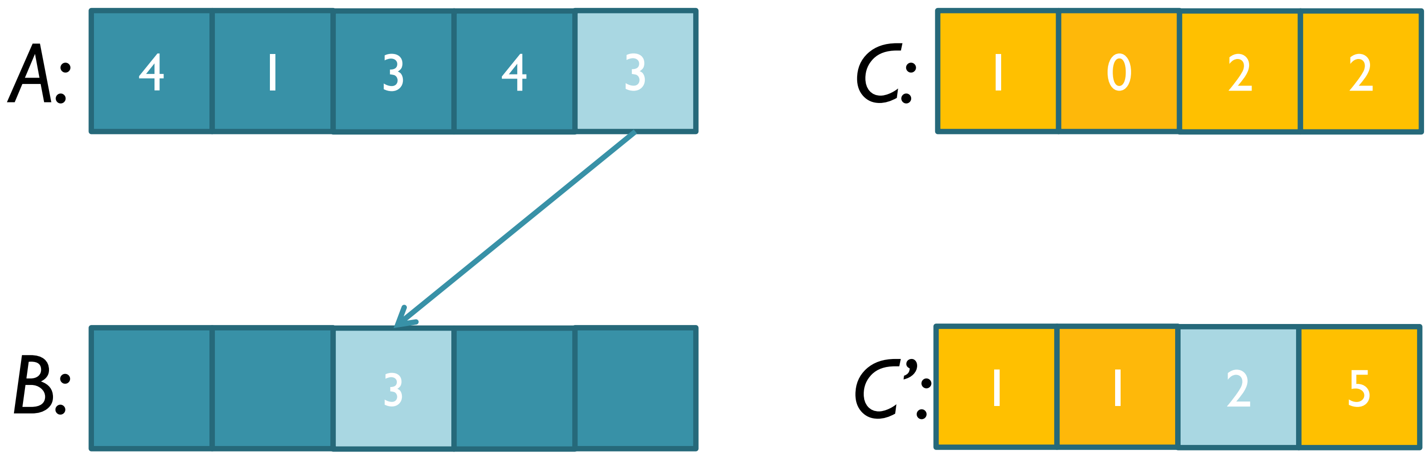
C':

1	1	3	5

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i-1]$

Loop 4

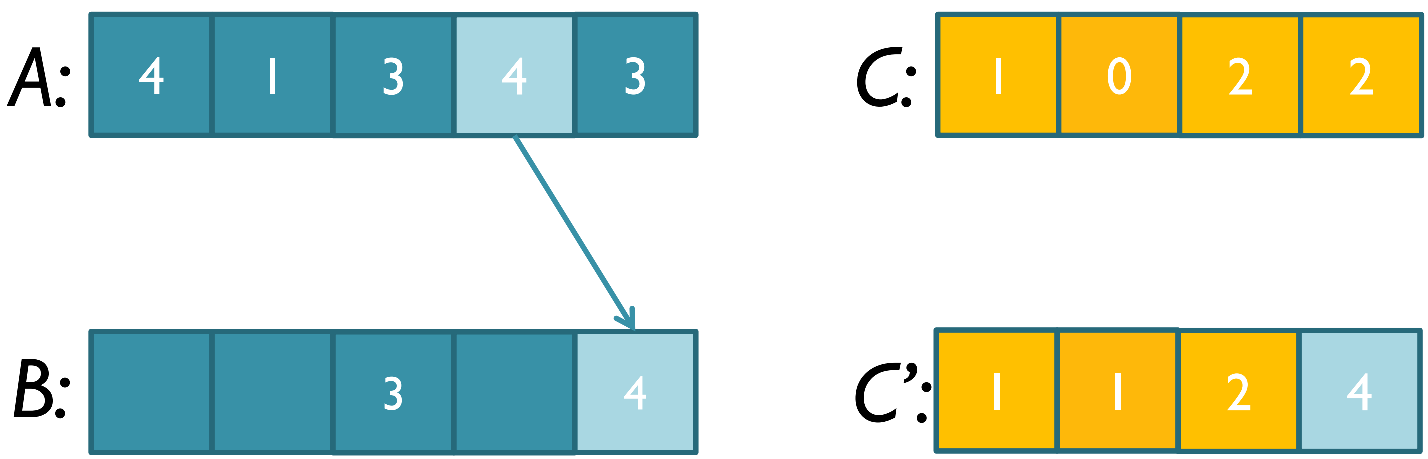


for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

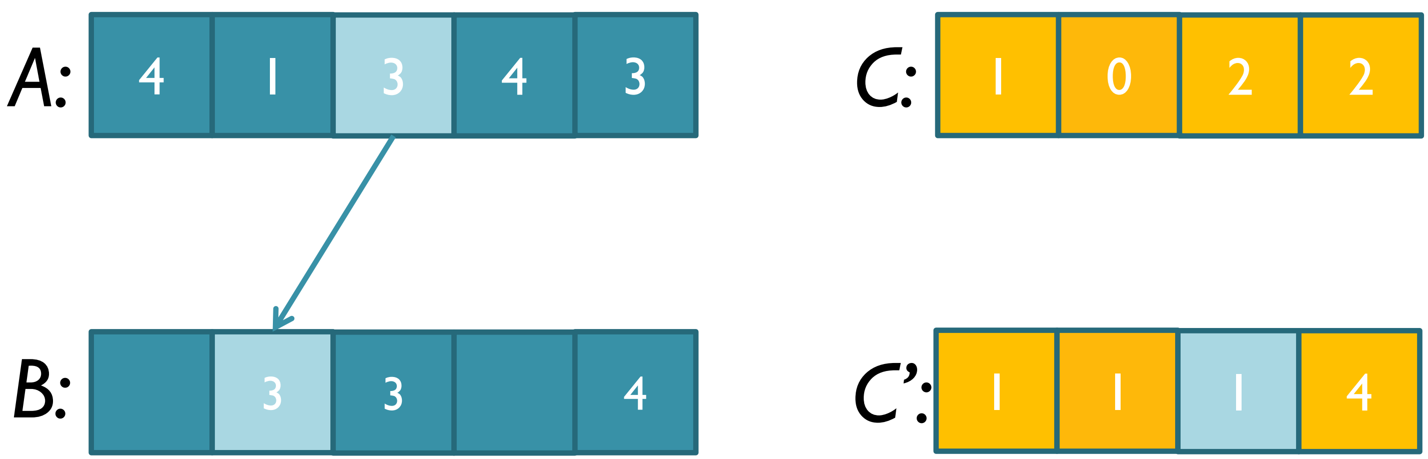


for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4

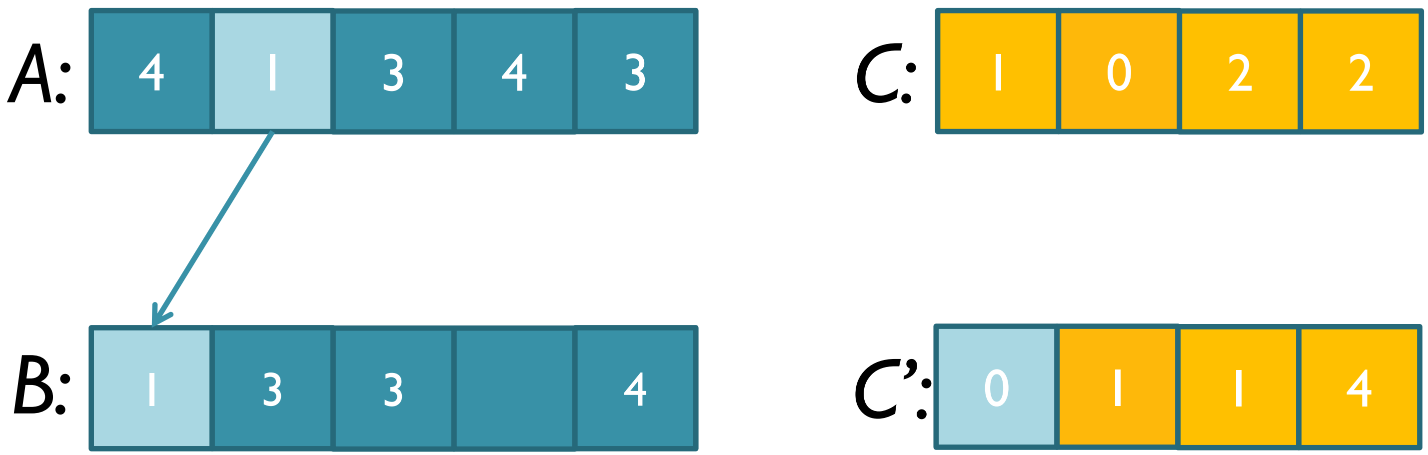


for $j \leftarrow n$ **downto** 1

  do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

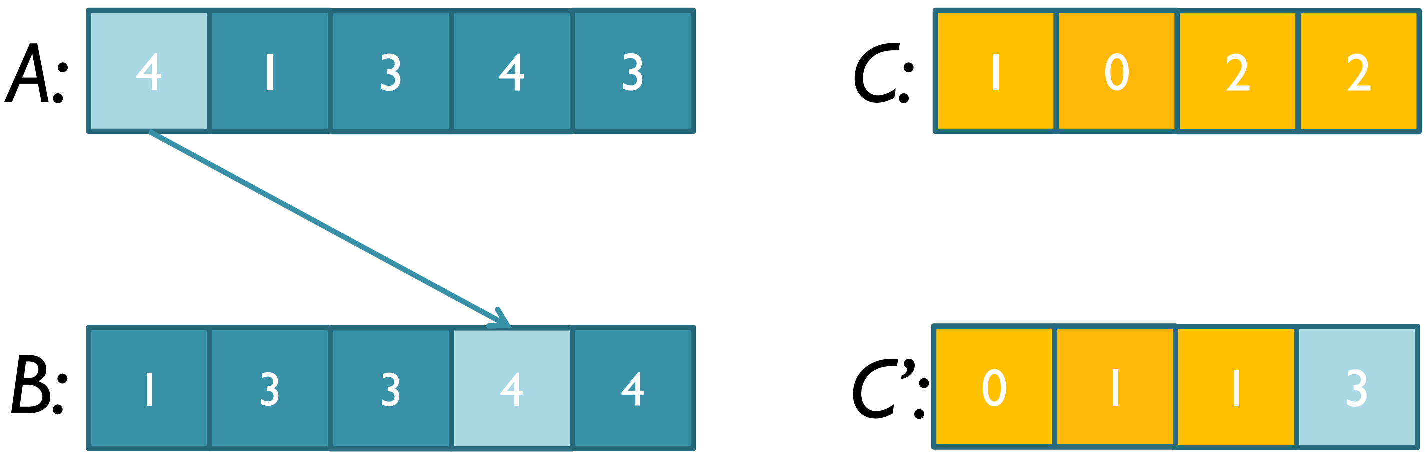
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



for $j \leftarrow n$ **downto** 1
   do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
      $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 4



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Complexidade

$$\Theta(k) \left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } k \\ \quad \textbf{do } C[i] \leftarrow 0 \end{array} \right.$$

$$\Theta(n) \left\{ \begin{array}{l} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \quad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{array} \right.$$

$$\Theta(k) \left\{ \begin{array}{l} \textbf{for } i \leftarrow 2 \textbf{ to } k \\ \quad \textbf{do } C[i] \leftarrow C[i] + C[i\text{–}1] \end{array} \right.$$

$$\Theta(n) \left\{ \begin{array}{l} \textbf{for } j \leftarrow n \textbf{ downto } 1 \\ \quad \textbf{do } B[C[A[j]]] \leftarrow \mathrm{A}[j] \\ \qquad C[A[j]] \leftarrow C[A[j]] - 1 \end{array} \right.$$

$$\Theta(n + k)$$

Complexidade

- Se $k = O(n)$, então o counting sort leva tempo $O(n)$
- Mas, ordenação é $\Omega(n \log n)$!
- Onde está o erro no nosso raciocínio?

Resposta:

- Counting sort não é um algoritmo baseado em comparações.
- De fato, não há qualquer comparação no algoritmo.

Seleção

- Problema de encontrar o *k*-ésimo menor (maior) elemento de uma sequência dada.
- É fácil determinar o menor elemento e/ou o maior elemento em tempo linear (como?)
- O algoritmo a seguir obtem o *k*-ésimo elemento em tempo linear no caso médio
- A lógica do algoritmo é parecida com a do algoritmo quicksort, por isso ele é chamado de *quickselect*

Quickselect

Function $\mathit{select}(s : \mathit{Sequence}\ \textbf{of}\ \mathit{Element}; k : \mathbb{N}) : \mathit{Element}$

$\quad$ **assert** $|s| \geq k$

$\quad$ pick $p \in s$ uniformly at random

$\quad$ $a := \langle e \in s : e < p \rangle$

$\quad$ **if** $|a| \geq k$ **then return** $\mathit{select}(a, k)$

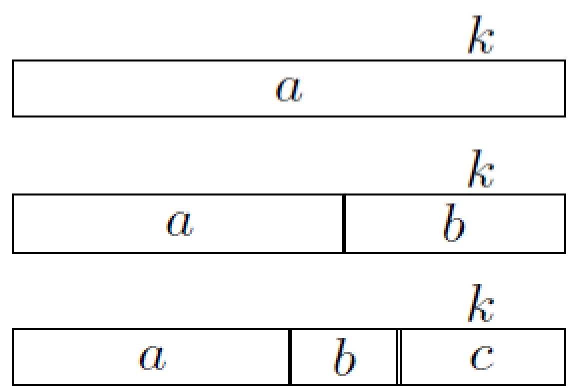
$\quad$ $b := \langle e \in s : e = p \rangle$

$\quad$ **if** $|a| + |b| \geq k$ **then return** p

$\quad$ $c := \langle e \in s : e > p \rangle$

$\quad$ **return** $\mathit{select}(c, k - |a| - |b|)$

Três situações



Exemplo

s	k	p	a	b	c
$\langle 3, 1, 4, 5, 9, \mathbf{2}, 6, 5, 3, 5, 8 \rangle$	6	2	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle 3, 4, 5, 9, 6, 5, 3, 5, 8 \rangle$
$\langle 3, 4, 5, 9, \mathbf{6}, 5, 3, 5, 8 \rangle$	4	6	$\langle 3, 4, 5, 5, 3, 4 \rangle$	$\langle 6 \rangle$	$\langle 9, 8 \rangle$
$\langle 3, 4, \mathbf{5}, 5, 3, 5 \rangle$	4	5	$\langle 3, 4, 3 \rangle$	$\langle 5, 5, 5 \rangle$	$\langle \rangle$

- Assim como o *quicksort*, o algoritmo *quickselect* tem complexidade de pior caso $O(n^2)$

Fim do módulo 4